

Coding For the Long Haul With Managed Metadata and Process Parameters

Mike Molter, d-Wise Technologies, Raleigh, NC

ABSTRACT

How robust is your SAS® code? Put another way, as you look through your program, how sensitive is it to changing circumstances? How much code is affected when, for example, the names of data sets to be analyzed or the names of variable within those data sets change? How are those changes expected to be implemented? In this paper we discuss program optimization and parameter management through the use of metadata. In the wide open, free-text environment of Base SAS, we too often worry more about getting results out the door than producing code that will stand the test of time. We'll learn in this paper how to identify process parameters and discuss programming alternatives that allow us to manage them without having to touch core code. We'll look at SAS metadata tools such as SQL Dictionary tables and PROC CONTENTS, as well as tools for reading and processing metadata, such as CALL EXECUTE. Finally, within our industry, we'll take a brief look at how the Clinical Standards Toolkit puts these methods into practice for CDISC compliance checking. This paper is intended for intermediate-level Base SAS users.

INTRODUCTION

As SAS programmers, the first thoughts that enter many of our minds when we hear the word "parameter" are those pertaining to macros. Macro parameters are the pieces of information that a macro collects from a user that allow program code to be generated and executed under a variety of circumstances. In this paper, however, we use the term in a broader sense. Simply stated, in this paper, parameters are the input to a given task or process. To the extent that such tasks are carried out by SAS code, parameters are also input to SAS programs. Whether or not you've given much thought to this way of dissecting a program's code, every program has them. Every string of text that collectively makes up the code in every program can be categorized in one of two ways - either SAS keywords and syntax (e.g. "FREQ", "SUBSTR", "OUTPUT", "=", ",", etc.), or the parameters or input upon which SAS syntax operates. As an example, consider the code excerpt in Program 1 below.

Program 1:

```
libname sdtm "my documents/mysdtm" ;  
proc freq data=sdtm.lb ;  
tables lbtestcd ;  
run ;
```

In Program 1, the underlined text represents the values of the program parameters. The first parameter is the location of the data, the second is the name of the data set, and the third is the name of a variable whose frequency we wish to analyze. In theory, it's the values of the program parameters that users can change while keeping everything else intact without affecting the fundamental meaning and purpose of the program. In practice, not all programs are written for changing circumstances. Some programs are only written for one-time use, while others, though they may be executed multiple times, may have no need to change parameter values. In such circumstances, identifying parameters may not be a high priority. In this paper,

however, we address the opposite scenario: under circumstances in which program code is meant to be executed under multiple circumstances with changing parameter values, how do we write program code that manages these parameters in an optimal way?

PARAMETER MANAGEMENT AND “HARD” AND “SOFT” PARAMETERS

The most primitive approach to handling parameter value changes is to allow users to directly edit the code. In Program 1 above users are allowed to open the program, navigate to the code that represents the parameter value they wish to change, and change it. This approach, however, is far from being described as “parameter management”. This situation addresses the concept of program access, and its importance can depend on factors such as how much code is accessible and to how many users is it accessible. If I am the only user being asked to generate these frequencies and there isn’t much more to the code in Program 1, then the consequences of my having to browse through the code and change parameter values are minimal. However, as the complexity, the scope, and the number of users increases, then it’s in everyone’s best interest to keep users away from *core code* in an effort to maintain the integrity and consistency of its purpose.

Keeping core code away from users generally means replacing specific instances of parameter values in the program with parameter references whose values are initialized either in the beginning of the program or in a separate program. For many SAS users this means an automatic call to the macro facility. Program 2 below provides the same functionality as Program 1, but is re-arranged in this manner.

Program 2

```
%let path = my documents/mysdtm ;
%let domain = lb ;
%let variable = lbtestcd ;
libname sdtm "&path" ;
proc freq data=sdtm.&domain ;
tables &variable ;
run ;
```

In this case parameters values are specified at the top of the program, but alternatively, the program author could have placed these specifications in a separate program that includes %include after the specifications to invoke the core code.

Of course when macro variables are the subject of conditional statement execution, iterative processing, or other types of more advanced code generation logic, programmers will write parameterized macros. These keep users away from core code in a way similar to that in Program 2 - by having users supply parameter values with the call to the macro. Program 3a is a macro that allows users to specify any number of variables from any number of data sets within a chosen directory. Each domain/variable combination is meant to be specified in the form *domain.variable*.

Program 3a

```
%macro freq(path=, domains=) ;
libname sdtm "&path" ;
/* Determine how many domains are requested */
%let domainnbr=%sysfunc(countw(&domains)) ;
/* Loop through the domain/variable list */
%do i=1 %to &domainnbr ;
/* Parse the ith domain/variable in the list */
%let domain=%scan(%scan(&domain,&i,%str( )),1,%str(.)) ;
```

```
%let variable=%scan(%scan(&domain,&i,%str( )),2,%str(.)) ;
proc freq data=sdtm.&domain ;
  tables &variable ;
run;
%end;
%mend ;
```

Because macros are compiled programs in catalogs and with SAS tools such as the autocall facility, it's easy for users to call macros at a safe distance from the core macro code. Program 3b illustrates a call to the macro defined in Program 3a.

Program 3b

```
%freq(path=my documents/mysdtm, domains=lb.lbtestcd eg.egtestcd vs.vstestcd) ;
```

Up to this point, if you have any experience at all with macros or even just macro variables, you haven't seen anything new. But this paper isn't about the macro facility. Sometimes the way a parameter value is specified for a task determines the best way to write code for it. Sometimes, even when the code for a parameterized task is expected to be executed multiple times on multiple occasions by multiple users who need to be kept away from the core code, the macro facility may not be the necessary solution. This brings us to the distinction between *hard* and *soft* parameter specifications.

In this paper, a hard parameter specification or value is one in which the parameter value(s) is explicitly stated. A soft specification, on the other hand, is less specific and more of a description of the input. For an example of each, let's examine two ways in which frequencies are requested.

Hard Parameter Specification

Please provide frequencies of LB.LBTESTCD, EG.EGTESTCD, and VS.VSTESTCD from the domains in the "my documents" directory.

Whether this represents a one-time request for frequencies or one in a series of requests in which the domains and variables change, the values of these parameters are stated explicitly. For purposes of this paper, let's imagine the latter case. The "hard" nature of the parameter specification leads us to three conclusions.

1. Each time such an analysis is needed, the wording of the request changes to reflect the specific variables needing analysis.
2. Carrying out the analyses with code means translating the request into the appropriate parameter specifications in code, which means some level of effort by the user to receive these values and insert them into the code.
3. To minimize the effort in #2 above as well as maintain consistency, the core code needs to be kept separate from the parameter specifications

What's key here is the nature of the parameter specification and in particular, the way that specification is "captured" and implemented in code. Now imagine that the following task is expected to be executed regularly.

Soft Parameter Specification

Please provide frequencies of all --TESTCD variables in the "my documents" directory.

Furthermore, let's assume that this request will never change, but what may change is the contents of the directory. We might say that even though the wording of the request remains

the same, parameter values are still changing implicitly because the word “all” encapsulates something different at each execution. The “soft” nature of the specification also leads us to three conclusions, two of which are stated here.

1. The wording of the request remains the same across executions
2. Carrying out the analyses with code still means translating the request into code, but the implicit nature of the specification requires more effort to capture parameter values.

Contrast these with the first two conclusions about hard parameter specifications. The extra effort described in #2 above is the effort needed by the user to translate the soft specification into a hard one at a point in time. In this case, the user must determine at the time of execution what variables in what domains match the description, and make sure all of the items on this list are analyzed. Programmers may see the changing nature of such a request and consider a macro in an effort to introduce consistency and reduce programmer error in writing the code, but let’s ask ourselves just how much error would be reduced. Any analysis and the code that supports it is only as good as the parameter values we supply it and the method in which we supply them. Whether or not a macro is used, the user is still forced to spend effort determining the analysis variables. While the macro approach is no worse than any other approach and may add value to some aspects of this task, it does nothing to reduce the potential for error in determining parameter values.

So where are parameter values? How do they get translated into code? Hard parameter values often get to code by way of the user’s brain involving a translation of a “non-readable” request. Maybe your boss verbalizes her request for a frequency during a meeting, or maybe she’s more formal and records it onto a form. Either way, you’re forced to cognitively consume this input and translate your interpretation into code. Your code is static in the sense that no link exists between the request and the code. A new request requires a manual update to the code. At any given execution of the code, soft parameter values must be translated into hard values. The robustness of the code, your ability to keep users away from core code, and the ease with which these hard values are determined from a soft specification depends on how you choose to go about this translation. When translated manually, soft parameter values can be more burdensome than hard ones because of the extra cognitive translation required. However, sometimes these translations *already exist in data - metadata*. By taking advantage of this data, programmers can write code that not only automates code generation, but also automatically captures “readable” parameter specifications from metadata, thereby eliminating the need for the manual cognitive translation while at the same time keeping users away from core code and maintaining a link between the request and the code. This leads us to our third conclusion about soft specifications.

3. To minimize the effort in #2, we write code that captures parameter specifications from metadata.

The remainder of this paper is divided into three parts - Tools, Examples, and a case study. In the Tools section, we’ll examine what SAS gives us for metadata. We’ll also discuss programming tools that help us to use this metadata. In Examples, we’ll start to see how these tools can be applied to the capture of soft parameter specifications. Finally, we’ll look at SAS’s Clinical Standards Toolkit (CST) as a case study. In this section, we’ll see that the development of a parameterized process doesn’t have to depend only on the metadata discussed in the Tools section. On the contrary, we’ll see a framework for CDISC compliance checking of submission data that offers compliance checking parameters through a well-planned, carefully designed set of metadata.

TOOLS

METADATA SOURCES

SAS has two main sources of metadata that describe data in your system. Maybe the more well-known of the two comes from the CONTENTS procedure. When the procedure code includes only the name of the data set without other options, the output contains a combination of data set- and variable-level metadata. At the data set level, this includes number of observations and variables, path and file name, and creation date. At the variable level, the name, type, and length of each variable is provided. Most noteworthy under these circumstances though is the fact that this information is provided by way of an output file (e.g. pdf, html, or other ODS output destinations). This is an acceptable solution when a user needs to cognitively process metadata but isn't ideal when metadata needs to be read or captured programmatically.

Figure 1 - Output from the CONTENTS procedure

Data Set Name	SASHELP.CLASS	Observations	19
Member Type	DATA	Variables	5
Engine	V9	Indexes	0
Created	Tuesday, May 24, 2011 02:40:19 PM	Observation Length	40
Last Modified	Tuesday, May 24, 2011 02:40:19 PM	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label	Student Data		
Data Representation	WINDOWS_64		
Encoding	us-ascii ASCII (ANSI)		

Engine/Host Dependent Information	
Data Set Page Size	4096
Number of Data Set Pages	1
First Data Page	1
Max Obs per Page	101
Obs in First Data Page	19
Number of Data Set Repairs	0
Filename	C:\Program Files\SASHome\SASFoundation\9.3\core\sasHELP\class.sas7bdat
Release Created	9.0301M0
Host Created	X64_PRO

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
3	Age	Num	8
4	Height	Num	8
1	Name	Char	8
2	Sex	Char	1
5	Weight	Num	8

We mentioned earlier that soft parameter values can be automatically generated when they can be captured from readable metadata. For that reason, the CONTENTS procedure through the OUT= option on the PROC statement gives users an opportunity to dump metadata to a data set. This data set contains one record per combination of LIBNAME (libref), MEMNAME (data set), and NAME (variable) and is sorted in this order.

Figure 2 - Partial illustration of the data set produced by PROC CONTENTS

	LIBNAME	MEMNAME	MEMLABEL	TYPMEM	NAME	TYPE	LENGTH	VARNUM	LABEL	FORMAT	FORMATL	FORMATD	INFORMAT	INFORML	INFORMD	JUS
1	SASHELP	CLASS	Student Data		Age	1	8	3			0	0		0	0	0
2	SASHELP	CLASS	Student Data		Height	1	8	4			0	0		0	0	0
3	SASHELP	CLASS	Student Data		Name	2	8	1			0	0		0	0	0
4	SASHELP	CLASS	Student Data		Sex	2	1	2			0	0		0	0	0
5	SASHELP	CLASS	Student Data		Weight	1	8	5			0	0		0	0	0

The DATASETS procedure, among its other capabilities, offers the same functionality as the CONTENTS procedure through the CONTENTS statement. Among the options available on this statement are DATA= and OUT= which serve the same purpose they do on the PROC statement of the CONTENTS procedure. The only difference is that in the DATASETS procedure, the data set named in the DATA= option is a one-level data set name and is assumed to reside in the library named in the LIBRARY= option on the PROC statement, whereas the DATA= option on the PROC statement of the CONTENTS procedure can be a one- or two-level data set name.

Whether or not an output data set is requested, the metadata produced by these two procedures is only generated when the procedures are executed, and only reflects the data requested in the procedures. The *dictionary tables* and *dictionary views*, on the other hand, always exist, are updated automatically whenever SAS files are added, deleted, or modified in any way, and reflect all data that reside in directories to which a libref has been assigned. The dictionary views are found in the library associated with the *sashelp* libref. Examples include VTABLES and VCOLUMN, which document table- and variable-level metadata respectively. VFORMAT and VMACRO contain information about formats and macros respectively, while others such as VOPTION and VLIBNAME contain other information about current SAS session settings. Dictionary tables, on the other hand, are available only through the SQL procedure, using the special *dictionary* libref. Many of these tables correspond to the *sashelp* views. For example, dictionary.tables has much of the same information as sashelp.vtable, and dictionary.columns overlaps with sashelp.vcolumn. Metadata about each of the dictionary tables can be found in the view sashelp.vdctry.

Just as the CONTENTS and DATASETS procedures can send to an output data set information about selected data sets and variables, the FORMAT procedure can send information about selected formats to a data set. By specifying a libref and catalog name with the LIBRARY= option on the PROC statement, a data set name on the same statement with the CNTLOUT= option, and optionally, the name of one or more formats with the SELECT statement, users can capture metadata about each entry in a format or informat.

Program 4

```
proc format library=fmtlib.mycatlg cntlout=fmtoutput ;
select fmtname ;
run;
```

Figure 3 - sample data set produced with CNTLOUT= option on PROC FORMAT

	FMTNAME	START	END	LABEL	MIN	MAX	DEFAULT	LENGTH	FUZZ	PREFIX	MULT	FILL	NOEDIT	TYPE	SEXCL	EEXCL
1	MYFMT	1	1	a	1	40	1	1	1E-12		0		0	N	N	N
2	MYFMT	2	2	b	1	40	1	1	1E-12		0		0	N	N	N
3	MYFMT	3	3	c	1	40	1	1	1E-12		0		0	N	N	N
4	MYFMT	4	6	d	1	40	1	1	1E-12		0		0	N	N	N
5	OTHFMT	10	10	100	1	40	3	3	1E-12		0		0	N	N	N
6	OTHFMT	11	11	101	1	40	3	3	1E-12		0		0	N	N	N
7	OTHFMT	12	12	102	1	40	3	3	1E-12		0		0	N	N	N

CALL EXECUTE - A TOOL FOR USING METADATA

We've now seen that SAS provides us with multiple sources of metadata. The next question is how to capture that metadata and generate code from it without going through the manual, error-prone method of typing code based on what we see in these data sets. For this we have CALL EXECUTE.

CALL EXECUTE is an executable DATA step statement whose purpose is to build code across DATA step iterations to be executed after the DATA step has finished executing. Because it's executable, it can be executed conditionally or iteratively. Its single argument is a valid SAS expression whose resolution yields the code to be executed. The expression can be a combination of literal text enclosed in quotation marks and references to variables (or functions of variables) found in the data set being read. References to these variables resolve to their value for the observation being read. Let's look at a few examples.

Program 5

```
data _null_ ;
  call execute("data x; set y; where z=1; run;") ;
run;
```

Program 5 is a trivial example that works fine but doesn't need the CALL EXECUTE. The DATA step doesn't actually read a data set. It simply generates a new DATA step that creates a data set called X by reading a data set called Y - a task that can be accomplished without CALL EXECUTE. However, suppose we only want to generate the DATA step that creates X if we know that Y exists.

Program 6

```
data _null_ ;
  if exist("y") then call execute("data x; set y; where z=1; run;") ;
  else put "SORRY, Y doesn't exist!" ;
run;
```

A more manual way to accomplish this task would have been for us to go searching through our directory structure for Y. If we find it, we execute code that creates X, if we don't, we don't. Perhaps we comment the code or even delete it. If it's already commented and we find the data set, then perhaps we un-comment it. Either way, this is a manual step we would have to take every time we want to execute this task. In Program 6, we use the EXIST function to determine if Y exists, and then generate the DATA step when it does. When it doesn't, we send a message to the log. In effect, by telling us whether or not a data set exists, the EXIST function is providing us metadata. By invoking it, we've accomplished some of the goals stated above. Namely, we've written code that automatically taps into metadata and executes code according to what it finds. This saves us from manually looking for this metadata and taking action based on our findings. It's code that can remain constant, even as the data set Y appears and disappears, without any need to manually wrap it in comments or remove comments.

Program 6 is still somewhat trivial in the sense that the DATA step still isn't reading any data and so it still isn't generating code based on any data it reads. In Example 7 we have a series of data sets that each needs to be sorted, each by its own unique set of sorting variables. Perhaps the most intuitive way to accomplish this task is to simply start writing a series of PROC SORTs. Assuming that no errors are made in the specification, interpretation, or translation into code of the parameter values (data set names, names of the sorting variables for each data set, etc.), this approach should be fine. But what happens when the same request is made at a later time with different parameter values? Maybe you make changes to your code (e.g. additional PROC SORTs, commenting or deleting unnecessary PROC SORTs, etc.)

according to the changes in parameter values. But once again, we see two problems here. The first is the manual intervention, both in terms of consuming parameter values and turning them into code, required each time the task is to be carried out. The second is the inability to have code robust enough to be linked to the request, regardless of parameter values, thereby forcing changes to core code.

Now let's imagine that parameter values for this request and others like it are kept in a data set called TABLES, as in Figure 4 below.

Figure 4 - TABLES.sas7bdat

	domain	sortvars
1	AE	USUBJID AESTDTC
2	DM	USUBJID
3	LB	USUBJID VISITNUM LBTPNUM LBTESTCD

Rather than manually typing each of the three PROC SORTs, we can use CALL EXECUTE to generate them.

Program 7

```
data _null_ ;
  set tables ;
  call execute("proc sort data=" || domain || " out=sorted_" || left(domain) || "; by
" sortvars ||";          run;");
run;
```

Unlike the previous two examples, Program 7 is generating code with every observation it reads from the data set TABLES. In particular, as it reads an observation, it generates a PROC SORT in which the value of the DOMAIN variable from TABLES becomes the value of the DATA= option on the PROC statement generated. Note also that an output data set is created by generating an OUT= option whose value is a concatenation of the literal text "sorted_" and the value of DOMAIN. The value of SORTVARS is plugged into the BY statement that's being generated.

Once again, central to our theme, this is code that works without any manual intervention in the core code, regardless of the parameter values, thanks to the stability and the manner in which they are captured. Earlier, we talked about the possibility of using output from SAS procedures such as CONTENTS or metadata from SAS dictionary tables as a source of metadata and code generation. In this case, rather than counting on what SAS makes available, we chose to store parameter values in a data set. In other words, rather than burying them in core code, we have pulled them out and put them into a separate file to which we grant user access. Here we show them in a data set, but we can imagine that the data set was built from input into a simple interface. For example, users who want to sort data sets might enter the name of the data set along with the sorting variables into an Excel spreadsheet that, after the push of a button, is converted into the TABLES data set.

It's also important to note that we haven't completely eliminated the need for some kind of manual intervention. In this case, that wasn't possible. But what we did accomplish was the transfer of that intervention from core code to an external file meant for public user access.

Let's now turn to a more clinical example. Those familiar with SDTM know that each class of domains has a set of variables that members of its class are allowed to contain. If other

variables containing information related to the domain are collected, then the values of those variables are put into a *suppqual* data set. What's peculiar about the Suppqual is that in order to keep the number of variables under control, the Suppqual has a vertical structure. So when variables are moved from a parent domain to a Suppqual, they aren't moved as individual variables, but rather, as values of one variable (QVAL) with a separate variable (QNAM) to keep track of the original variable names. One of the requirements of any observation in a Suppqual data set is that it contains information that links it back to the record(s) in the parent domain from which it came. These parent domain records are identified by IDVAR and IDVARVAL.

Figure 5
ae.sas7bdat

	usubjid	aeseq	aegrpid	aeterm	aestdtc	aesev
1	123	1	A1	Tummy ache	2013-03-24	MILD
2	123	2	A1	Tummy ache	2013-03-24	MODERATE

suppae.sas7bdat

	studyid	rdomain	usubjid	idvar	idvarval	qnam	qval
1	ABC	AE	123	AEGRPID	A1	SUPP1	VAL1
2	ABC	AE	123	AEGRPID	A1	SUPP2	VAL2
3	ABC	AE	123	AESEQ	1	SUPP3	VAL3
4	ABC	AE	123	AESEQ	2	SUPP3	VAL4

Figure 5 shows a small example of an AE data set along with its corresponding SUPPAE. From the SUPPAE we find three variables - SUPP1, SUPP2, and SUPP3 - that have been collected with other adverse event data but that didn't have a place in the parent AE data set. From the combination of values in IDVAR and IDVARVAL, we can see that the value VAL1 of SUPP1 and VAL2 of SUPP2 tie back to both observations in AE. We also see that the value VAL3 of SUPP3 ties back to record 1 in AE, and the value VAL4 of SUPP3 ties back to record 2.

To the extent that these supplemental variables are needed for analysis, it's often necessary to merge the supplemental information back in with the core data in the parent domain. In our next example we attempt to build sustainable code that uses metadata to generate code that merges supplemental data from SUPPAE into the parent domain.

For starters, because we want to bring the supplemental information back to the parent domain as variables, we'll use the TRANSPOSE procedure to transpose the vertical SUPPAE to a horizontal structure that we'll call SUPPAE_TRAN. The code for accomplishing this is left to the user but the data set is illustrated in Figure 6.

Figure 6 - suppae_tran.sas7bdat

	usubjid	idvar	idvarval	_NAME_	SUPP1	SUPP2	SUPP3
1	123	AEGRPID	A1	qval	VAL1	VAL2	
2	123	AESEQ	1	qval			VAL3
3	123	AESEQ	2	qval			VAL4

At first glance the merge may seem straightforward, but that perception changes when we consider the merging variables. What's unusual about Suppqual is that it contains data in the QVAL variable, but in IDVAR it contains metadata. In this example, IDVAR tells us that to merge SUPP1 and SUPP2 with the parent domain, we merge the values in IDVARVAL with

values in the parent domain's AEGRPID. To merge SUPP3, we match values of IDVARVAL with AESEQ in the parent domain. For that reason, we can merge SUPP1 and SUPP2 back with AE at the same time, but SUPP3 must be merged back separately.

Much like we might do when writing a complicated macro, let's paint a picture for ourselves of what the code we want to generate might look like.

```
proc sql ;
  create table ae1 as
  select a.*, b.supp1, b.supp2
  from ae a left join suppaetrans(where=(idvar eq 'AEGRPID')) b
  on a.usubjid=b.usubjid and a.aegrpid=b.idvarval ;

  create table ae2 as
  select a.*, b.supp3
  from ae1 a left join suppaetrans(where=(idvar eq 'AESEQ')) b
  on a.usubjid=b.usubjid and a.aeseq=input(b.idvarval,8.) ;
quit;
```

The choice of SQL over the DATA step was a choice of convenience. The DATA step could have been used but the SQL procedure doesn't require the data sets being merged to be sorted beforehand, and it allows for more complex merging conditions, including merging (or joining) by variables whose names aren't the same (note the second join condition in each query).

Now let's give some thought to the kind of metadata we'll need to generate this code. For starters, we notice that we have one SQL query for each merging variable. Each of these merging variables (i.e. values of IDVAR) is associated with a unique set of variables (i.e. values of QNAM) to bring into the parent domain. These are the variables with the "b." prefix on the SELECT line. Each of these queries is also creating a data set whose name begins with the name of the parent domain and is suffixed with a counter that increases by one with each query (e.g. AE1 is created by the first query, AE2 by the second, and so on). The final requirement that isn't so obvious is the Type attribute (i.e. character or numeric) of the variable named in IDVAR. We know that IDVARVAL is always character so when we join it with a numeric variable like AESEQ, we need to convert AESEQ to character or IDVARVAL to numeric. This is the reason for the INPUT function in the second query above. With some simple DATA step code, most of this metadata structure can be derived from SUPPAE. The TYPE values for each IDVAR value will have to be extracted from SAS metadata (e.g. dictionary tables or PROC CONTENTS output). Compare the data set META illustrated in Figure 7 with the original SUPPAE in Figure 6.

Figure 7 - meta.sas7bdat, to be used as parameter input for the task of merging supplemental AE data into the AE data set

	idvar	keepvars	counter	type
1	AEGRPID	SUPP1 SUPP2	1	C
2	AESEQ	SUPP3	2	N

We now have most of the information in META that we need to generate the code above, but given the way the values appear, we still have some work to do to generate the SQL code. For example, the values of KEEPVARs in META will have to turn into a comma-separated list and each item in the list will have to have "b." prepended to generate the SELECT line. The number found in COUNTER will have to be concatenated to the name of the parent domain to generate the name of the data set being generated, and a value of "N" in TYPE will have to generate the

INPUT function. With a little more data manipulation code in the DATA step that created META, we can generate a new and improved META2 that will make the CALL EXECUTE argument in the final DATA step much simpler.

Figure 8 - meta2.sas7bdat

	idvar	keepvars	joincond	dataset
1	AEGRPID	b.SUPP1,b.SUPP2	b.idvarval	AE1
2	AESEQ	b.SUPP3	input(b.idvarval,8.)	AE2

We're now ready to write the DATA step that generates the SQL code. We start by noticing that the PROC statement itself is generated only once at the beginning, and the QUIT statement is generated just once at the end. Proper generation of these statements requires the use of internal DATA step variables.

```
data _null_ ;
set meta2 end=thatsit ;
if _n_ eq 1 then call execute ('proc sql ;' ) ;

other code

if thatsit then call execute (' quit; ' ) ;
run;
```

Now let's consider the "other code." Using the PROC SQL above as a guide, we can see that with every record we read from META2, we want to generate a query in which the name of the data set being created is found in the DATASET variable of META2. The query will left join two tables. In the first query, the left table (the table in which all records are kept) is the original parent domain. In subsequent queries, it's the data set created in the previous query. The right table is always SUPPAE_TRAN subsetted down to only those records whose value for IDVAR is the value of IDVAR in the observation being read from META2. The variables being selected are everything from the left table, and only those from the right table that are in the current value of the KEEPVARs variable in META2. Finally, the join condition equates the current value of IDVAR from META2 with the expression in JOINCOND in META2. At this point, the only information we don't have in META2 is the name of the left data set (i.e. the data set created in the previous query), so we'll have to create it with the same DATA step that contains the CALL EXECUTE.

```
data _null_ ;
length lastds $4 ;
set meta2 end=thatsit ;
if _n_ eq 1 then do;
call execute ('proc sql; ' ) ;
lastds='ae';
end;
retain lastds ;

call execute ('create table '||data set||' as select a.*, '||left(keepvars)||' from
' ||lastds a left join suppae_tran(where=(idvar eq " '||compress(idvar)|| '")) b on
a.usubjid=b.usubjid and a.' ||compress(idvar)|| '=' ||left(joincond)|| ');

if thatsit then call execute(' quit; ' ) ;
lastds=data set ;
run;
```

Note that we've created a variable called LASTDS. When the first observation is read, this is assigned the value "ae". LASTDS is retained so when it's assigned the value of DATASET at the end of the DATA step, it keeps that value as it reads the next observation. This variable is then referenced as the FROM clause is generated.

SOLVING THE SOFT PARAMETER PROBLEM

We've now seen some of the ways that SAS stores system information and metadata. We've also seen that CALL EXECUTE is a tool that can read data through the DATA step and generate code from it. In this section we turn back to the problem of writing sustainable code based on soft parameter specifications. We'll start with an example that has several possible solutions. One involves CALL EXECUTE, the others involve unique ways of using familiar tools.

Example: Create a 0-observation data set called XYZ structured like the currently existing data set ABC

Let's first be clear on what "structured like" means. For purposes of this example, it means that XYZ should have variables of the same name, length, type, label, and format as those found in ABC. Additionally, the variables should be ordered in XYZ in the same way they are ordered in ABC.

Figure 9a: ABC.sas7bdat (variable labels shown as column headers)

	Name of Student	Gender	Age of student	Height on 1st day	Weight on 1st day
1	Alfred	Male	14	69	112.5
2	Alice	Female	13	56.5	84
3	Barbara	Female	13	65.3	98
4	Carol	Female	14	62.8	102.5
5	Henry	Male	14	63.5	102.5
6	James	Male	12	57.3	83
7	Jane	Female	12	59.8	84.5
8	Janet	Female	15	62.5	112.5
9	Jeffrey	Male	13	62.5	84
10	John	Male	12	59	99.5

Figure 9b: Output data set ABCCON.sas7bdat from PROC CONTENTS of ABC.sas7bdat

	LIBNAME	MEMNAME	NAME	TYPE	LENGTH	VARNUM	LABEL	FORMAT
1	WORK	ABC	Age	1	8	3	Age of student	
2	WORK	ABC	Height	1	8	4	Height on 1st day	
3	WORK	ABC	Name	2	8	1	Name of Student	
4	WORK	ABC	Sex	2	1	2	Gender	\$SEX
5	WORK	ABC	Weight	1	8	5	Weight on 1st day	

Before exploring our options, let's be clear on where the soft parameter specifications are in this problem. A hard specification might be worded something like "create a data set called XYZ whose first variable called NAME has a label of 'Name of Student,' whose length is 8, whose type is character", etc. Other variables and their attributes would also be explicitly stated. The current way, however, in which this problem is stated has a soft nature because it requires the user to determine what the variables and their attributes are.

Without tools, users might accomplish this task with one of any number of combinations of ATTRIB, LENGTH, FORMAT, and LABEL statements. More importantly though, from where

and how would they get the attribute values? In an interactive environment users might double-click on the ABC data set, then double-click on the variables that then opens a dialog box with this information. Or perhaps they would run PROC CONTENTS and allow for the default behavior where metadata is sent to a listing file. Either way, users would then transcribe it into their program. And then next month, when the same request is made, the user goes through the same exercise to see if anything has changed about ABC. The following methods save the user all of that work.

The most straightforward option is with the familiar SET statement. We sometimes think of this statement as more of an executable statement than a compile statement, but the fact is that it's both. At compile time, SAS will begin the creation of XYZ by defining its variables based on what it finds in ABC, the data set named in the SET statement. It's only when execution begins that it starts reading data from ABC, and it's at the end of a DATA step iteration when it creates an observation. If you don't want any observations created, the STOP statement will halt execution before observations are written.

```
data XYZ ;
set ABC ;
stop ;
run;
```

A second option is available through PROC SQL with the keyword LIKE on the CREATE TABLE statement.

```
proc sql ; create table XYZ like ABC ;
```

A third option is to use CALL EXECUTE in the DATA step to generate ATTRIB statements. As with any other code-generating task, let's first be sure we know what code we want to generate.

```
data XYZ ;
attrib NAME label='Name of Student' length=$8 ;
attrib SEX      label='Gender' length=$1 format=$sex. ;
attrib AGE      label='Age of student' length=8 ;
attrib HEIGHT label='Height on 1st day' length=8 ;
attrib WEIGHT label='Weight on 1st day' length=8 ;
stop;
run;
```

Before generating the code, let's make a few observations. First, as every DATA step programmer knows, the DATA statement appears once in the beginning and the RUN statement appears once at the end. As we did earlier, we'll want to generate each of these statements when the first and last observations of ABCCON are read. Second, by comparing the code we want to build to the data set ABCCON, it's clear that each observation read from ABCCON will generate an ATTRIB statement. Third, we note that the order of the ATTRIB statements determines the order of the variables in XYZ. For that reason, before reading ABCCON, we'll want to make sure it's sorted by VARNUM.

```
proc sort data=abccon ;
by varnum ;
run;
```

Fourth, keep in mind that the specification of length includes a dollar sign for character variables. And finally, fifth, we notice that when the FORMAT variable in ABCCON is null, we don't generate "format=" text.

```

data _null_ ;
set abcccon end=thatsit ;
if _n_ eq 1 then call execute(' data xyz ; ' ) ;

```

Once again, we use the END= option on the SET statement that allows us to identify the last observation being read from ABCCON so that we have a condition upon which we can base the generation of the STOP and RUN statements. We'll now generate the ATTRIB statement and the LABEL= option. Keep in mind that quotation marks are required to surround the label. Here we're using single quotes to surround literal text arguments of CALL EXECUTE, so we'll generate double quotes to surround the label.

```

call execute(' attrib '||compress(name)||' label="'||trim(left(label))||'" ' ) ;

```

Now we'll generate the LENGTH= attribute, but we'll generate the dollar sign when the variable whose attribute we're defining is character.

```

if type eq 1 then call execute(' length='||compress(put(length,best.)) ) ;
else call execute(' length=$'||compress(put(length,best.)) ) ;

```

Now we'll generate a FORMAT= option if a format is attached to the variable.

```

if not missing(format) then call execute(' format='||compress(format)||'.') ;

```

Keep in mind that the last the last three steps have all been generating different options on the same ATTRIB statement. We're now finally ready to generate the semicolon that ends this statement.

```

call execute(';') ;

```

And finally the end of the DATA step.

```

if thatsit then call execute('stop; run;') ;

```

In our next example, we return to a problem stated early on this paper.

Example: Provide frequencies for all --TESTCD variables in the My Documents directory.

Assuming we have assigned the libref *mysdtm* to this directory, we first use PROC CONTENTS to produce a data set that tells us the names of all --TESTCD variables (NAME) and in which data sets they live (MEMNAME) in this directory.

```

proc contents data=mysdtm._all_ noprint out=contents(where=(index(name,'TESTCD') gt
0)) ;

```

Knowing that any data set can contain only one such variable, each observation read should generate its own PROC FREQ.

```

data _null_ ;
set contents ;
call execute('proc freq data=sdtm.'||compress(memname)||'; tables
' ||compress(name)||'; run; ' ) ;
run;

```

In our final example, we address a problem of interest to SDTM users. The SDTM structure comes with a set of rules that are laid out in an “unreadable” PDF-formatted implementation guideline (SDTMIG) document. Being unreadable means that without intelligent metadata, programmers wishing to programmatically check the compliance of their SDTM data must cognitively capture what is subject to those rules and translate into code. Sometimes the metadata is something available from the output of PROC CONTENTS or dictionary tables, but not always. When it’s not, then defining custom metadata in readable SAS data sets allows for the possibility of code generation in ways we’ve discussed up to this point.

For example, SDTM has the concept of “required” variables. Variables that are deemed required must be present and must never have null values. Examples include the variable USUBJID in all subject-related domains, VSTESTCD from VS, AETERM and AESEV from the AE domain, and many more. Writing code to check that VSTESTCD (and other required variables) exists and is always populated should be relatively straightforward. What we’re concerned with here is how we get VSTESTCD into the list of variables to be checked in the first place.

Example: Identify all required variables that are not present or that have null values.

The soft specification here is “all required variables.” The user is left with determining what these are and then implementing the rule for each. The SDTMIG provides tables that tell us which variables are required, so again, we can’t get away from some minimal amount of manual intervention in translating from PDF to SAS. The question is how.

Of course we can always take the straightforward code we mentioned above for VSTESTCD and simply repeat it, making sure to change data set and variable name references, but as we’ve seen up to this point, this approach doesn’t always stand the test of time and change. Imagine if the SDTM standards changed so that more required variables are introduced, or that variables that in one version were not required are now required. Or imagine that an individual sponsor in a particular therapeutic area wants to require a variable that CDISC did not require. Once again, with this approach, changing circumstances require manual intervention inside the core of such code.

On the other hand, instead of program code with repeated blocks of the same kind of code that differ only by the parameter values, let’s imagine metadata that documents whether or not an SDTM variable is required or not. Of course such metadata is not available through dictionary tables or PROC CONTENTS, but there isn’t any reason we can’t create it ourselves. Figure 10 below illustrates a data set META.sas7bdat containing this information.

Figure 10

	dataset	variable	core
1	VS	USUBJID	Req
2	AE	AETERM	Req
3	AE	AESEV	Req
4	LB	LBCAT	Exp
5	VS	VSSTRES	Exp
6	CM	CMOCCUR	Perm

Again, because its root is in a PDF format, META will have to be created manually, and changes will also have to be made manually, but not to core code. Those responsible for

maintaining changes can do so through code that creates META, or better yet, through an interface (e.g. Excel) that creates it. Core code that uses META can then be left untouched.

```
proc sort data=mysdtm ; by memname name ; run;
proc sort data=meta (where=(core eq 'Req')) out=metasort; by data set variable ;
run;

/* Check for the existence of required variables */
proc contents data=sdtm._all_ out=mysdtm noprint ; run ;

data reqnotexist ;
merge metasort(in=m) mysdtm(in=mys rename=(memname=data set name=variable)) ;
by data set variable ;
if m and not mys ;
run;

/* Check for unpopulated required variables */
data _null_ ;
set metasort ;
by data set variable ;

if first.data set then call execute("data reqnull_" || compress(data set) ||"; "
||" set sdtm." || compress(data set) || ";"
|| " where missing (" || compress(name) || ")" ) ;

else call execute(" or missing(" || compress(name) || ")" ) ;

if last.data set then call execute(" ; run; ") ;
run;
```

Due to the approach we took, with parameter values being managed in META, the code above can remain the same even as these values change. The first DATA step simply merges METASORT (the sorted version of META with only required variables) with PROC CONTENTS output of the SDTM directory. Observations in METASORT that have no matches in this output equate to variables deemed through metadata as required that aren't in the data itself. The second DATA step uses *first.* and *last.* logic to generate DATA step code for each domain that has required variables. In particular, for each such domain, a data set whose name starts with REQNULL_ and ends with the domain name is created. Observations from the corresponding domain are put in this data set if any of the required variables have a missing value for that observation. This is achieved by generating code of the form

```
if missing(x1) or missing(x2) or missing(x3) etc ;
```

The tables in the SDTMIG contain other information about SDTM variables. Some of it is informational only, but some of it might be entered into META as additional variables that could be used for additional compliance checks. For example, the SDTMIG also provides the name of a codelist of allowable values for each applicable variable.

Figure 11

	dataset	variable	core	codelist
1	VS	USUBJID	Req	
2	AE	AETERM	Req	
3	AE	AESEV	Req	SEV
4	LB	LBCAT	Exp	LBCAT
5	VS	VSSTRES	Exp	UNIT
6	CM	CMOCCUR	Perm	NY

THE CLINICAL STANDARDS TOOLKIT - A CASE STUDY IN PARAMETER AND METADATA MANAGEMENT

On the surface, the Clinical Standards Toolkit (CST) is a set of SAS files available free of charge to any BASE SAS license, that is used to execute compliance checks against SDTM data and to build define.xml. The real defining characteristic of the CST, however, is its framework - the way that its files are used with each other. As we'll see here, this framework takes full advantage of the notion of parameter management.

For starters, SAS has done us the favor of creating the metadata suggested in our last example, and any other metadata needed for certain compliance checks. In other words, SAS has gone to the trouble of cognitively consuming rules and metadata found in the SDTMIG and organizing it into two SAS data sets referred to as *reference metadata* (one for table-level metadata and the other for variable-level). The content of these data sets includes metadata for all domains that are documented in detail in the SDTMIG, plus all variables these domains can contain. In reality, companies developing standards based on SDTM will not need all of these, and may also need additional custom domains. For this reason, it is expected that these will need customizing. Companies need to define processes for managing the content and the changing nature of these data sets but what SAS provides us in the CST is a good start.

Among other files that come installed with the CST are well over 100 macros. These macros and their relationship to each other are key to the operation of any CST process. A handful of these are dedicated to specific standards such as SDTM and ADaM. These macros are located in directories that live alongside directories that house the reference metadata (referred to in this paper as the *user area*). Although it shouldn't usually be necessary, placing these macros in this location with this kind of user visibility allows users to make custom modifications to them, and if desired, even combine them with user-written standard-specific macros for the sake of organization. Most CST macros, however, that are dedicated to general CST processes, are buried with other SAS installation files, often in hidden, or at least read-only locations that are not meant to be visible to users. These macros, sometimes referred to as framework macros, unlike the standard macros, automatically become part of the autocall facility and so are automatically available for use. While some of these are meant for direct use by users, many are meant only to be called within the call of other macros.

Considering all of the information that a compliance-checking process needs, it's easy to see that it is a highly parameterized process. Such information includes the location of the data to check, and which checks should be executed. We mentioned that SDTM rules are documented in reference metadata, but the process needs to know where this metadata is. Frequent users of the macro facility might then assume that the macro that kicks off the process is defined with parameters designed to collect this information from the user. These same users would then be surprised to find out that *sdm_validate*, the macro that initiates the validation of SDTM compliance, has no defined macro parameters.

CST STARTER DATA SETS

It takes an open SAS mind and a bit of imagination for SAS users to realize that process parameters don't have to be implemented through macro parameters, even if the process is highly driven by macros, as CST processes are. Up to this point in the paper we've introduced the notion of parameter management through well-planned metadata, documented in SAS data sets. If administrators of such metadata can collect these parameters from users in intuitive,

user-friendly ways, such as through interfaces, then they can give these users the flexibility to easily alter parameter specifications while at the same time, keep them away from core code. Many CST macros are defined with parameters, but the kickoff macros for CST validation processes, without parameters, force the same approach on CST users - define a small group of administrators to maintain metadata, macros, formats, and other utilities, and define a larger group to define parameter values and execute CST macros.

In addition to reference metadata, as part of the CST, SAS provides us with several other starter data sets. At the highest level outside of any specific standard context are two data sets called STANDARDS and STANDARDSASREFERENCES. These data sets contain information about the location of important files for each standard. A standard is said to be *registered with CST* if this information about the standard is in these two data sets (a procedure that uses a CST macro). The location of these two data sets is expected by certain macros and so cannot change, but directory structure for standards and their files is more flexible, as long as it's accurately reflected in these two data sets.

CST is installed with certain starter pre-registered standards, including a starter SDTM standard based on the SDTMIG. Each of these standards is accompanied by starter data sets and other files that each contain unique input to CST processes. Among these files is the reference metadata mentioned earlier, that contains table- and column-level metadata about all domains and their variables that are mentioned in SDTMIG.

Each standard also comes with a Properties file. Property files are simply text files with name-value pairs. Within a standard, these pairs represent standard-specific properties. The starter files contain default values. CST also comes installed with a special Framework standard which is mostly defined by its properties file which contains CST property assignments that are more global in nature.

Every standard that supports compliance checking also comes with a data set called VALIDATION_MASTER. This data set documents all the checks to which a standard is subject. Version 1.4 has more than 200 checks for SDTM data. Information about each check includes the severity of a violation (i.e. Info, Warning, Error), the source of the check (e.g. WebSDM, SAS, OpenCDISC), and the status of the check (e.g. ready for use, not ready). Some of this information is used only in reporting, but other information is read and used as a vital part of the checking process itself. For example, the values of CODESOURCE represent macro names. After some initial setup, kickoff macros like *sdtm_validate* call these macros. For certain checks, the variable CODELOGIC is populated with valid SAS code that certain macros expect to execute.

VALIDATION_MASTER is named the way it is because it represents a master list of all validation checks applicable to a process. Again, what SAS gives us is meant to be a starter file, but through time, administrators may discover that certain checks are never executed in their environment and so can be deleted. Many administrators, on the other hand, may have a need to execute custom compliance checks. To do this requires an intimate knowledge of how this file is used by *sdtm_validate* and other macros, as well as what *sdtm_validate* does with the results. Administrators should take the time to get comfortable with this and develop processes around the addition of custom checks to the master file.

Although in theory this data set has everything needed for a compliance checking process, in reality, it contains too many checks to execute all at one time. For that reason, best practice dictates that each execution only runs a subset of the master list of checks. Because the subset

list of checks to execute is itself, a parameter, the act of subsetting the master list must involve the creation of a new data set for the process to read.

Finally, one other way in which the process is parameterized is in the messages generated as a result of execution. As with properties, CST has framework messages as well as standard-specific messages. The former are documented in a MESSAGES data set in the framework standard. These are general messages that pertain not to specific compliance checking results, but more generally to results of program execution. Think of them as messages you might see in your log. In short, many of the CST macros try to capture other issues outside of compliance that may have come up in the job, and report on them with messages from this data set. To some extent, this saves users from having to browse through a long log for colored text for troubleshooting program execution. Standard-specific messages, on the other hand, do convey compliance-specific issues.

One final data set that all CST processes use is the SASREFERENCES data set. This data set documents the location of all files and directories important for the process. Many observations in this data set are pulled from the STANDARDSASREFERENCES mentioned earlier, such as location of reference metadata for the standard, and location of the data set containing the subset of VALIDATION_MASTER. In addition to these though, this data set may include study-specific locations such as the location of study data, or custom macro or format directories. In addition to directory and file locations, other variables in this data set represent filerefs and librefs, as well as the order in which multiple macro or format libraries are to be searched.

A CST process like compliance checking is kicked off with a macro like *sdtm_validate*, but prior to this macro call, the framework properties file and the SASREFERENCES file must be processed. Macro calls to process these files along with a kickoff macro like *sdtm_validate* are executed in a driver program. Processing the framework properties file simply means executing a macro that creates macro variables from the property names and assigns them the property values. One of the framework properties indicates the location of the SASREFERENCES file. With this information, another macro will read SASREFERENCES, execute FILENAME and LIBNAME statements according to the directory and file locations it finds, add macro libraries it finds to the autocall facility, and define a FMTSEARCH path for any format catalogs it finds. With all of this information now known to CST, it's ready to begin compliance checking.

As with any task that involves executing parameterized code, the user's main responsibility is to make sure that parameter values reflect the current circumstances. With this kind of approach, this equates to making sure that metadata content that holds parameter specifications is accurate. This is a form of manual intervention, and depending on how the metadata is created, may even involve some code intervention, but the core code that's at the heart of the task is left alone.

CONCLUSION

Every program we write reflects one or more parameterized tasks from which, one way or another, parameter values must be consumed and written into the program. The SAS language is written in a way that naturally mixes parameter values with core code. Under many circumstances this is ok but as parameter values change, other factors must be considered. We often want to reach for the macro facility in this case, but in this paper we considered a different kind of change in parameter values. Namely, we considered soft parameter values in which the wording of the specification doesn't change but the specifics of its meaning does. In some cases the meaning of the soft parameter could be found in SAS-provided metadata, but in

cases where it isn't, we discovered that we could gain the same benefits with carefully designed pre-defined metadata. Such benefits include the inclusion of code parts in our metadata, and the ability to use SAS tools to read this metadata and generate code. This has the effect of removing parameter values from core code and maintaining them in a separate location. With careful process planning, users have the flexibility to customize parameter values without any access to core code, thereby maintaining its integrity and sustainability.

CONTACT INFORMATION

Please feel free to contact me with comments or questions in any of the following ways:

Mike Molter
d-Wise Technologies
919-414-7736 (mobile)
919-600-6237 (office)
mike.molter@d-wise.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.